भारतीय विज्ञान संस्थान

# An Evaluation of Basic Protection Mechanisms in Financial Apps on Mobile Devices

| | |
|---|---|
| Journal: | *IISc - Masters Thesis Processing* |
| Thesis ID | masters-2022-0022 |
| Manuscript Type: | Synopsis and Thesis |
| Date Submitted by the Author: | 29-Apr-2022 |
| Complete List of Authors: | Agrawal, Nikhil; Indian Institute of Science, CSA; |
| Keywords: | Android Financial Apps, Reverse Engineering, OWASP |
| | |

SCHOLARONE™
Manuscripts

# An Evaluation of Basic Protection Mechanisms in Financial Apps on Mobile Devices

A THESIS

SUBMITTED FOR THE DEGREE OF

**Master of Technology (Research)**

IN THE

**Faculty of Engineering**

BY

**Nikhil Agrawal**


Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

April, 2022

For Review Only

# Declaration of Originality

I, **Nikhil Agrawal**, with SR No. **04-04-00-10-22-19-1-16620** hereby declare that the material presented in the thesis titled

**An Evaluation of Basic Protection Mechanisms in Financial Apps on Mobile Devices**

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2019-2022**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name(s):

Prof. Kanchi Gopinath Advisor Signature

Prof. Vinod Ganapathy Advisor Signature



For Review Only

For Review Only

© Nikhil Agrawal
April, 2022
All rights reserved

For Review Only

For Review Only

DEDICATED TO

*My Parents and My Brother*

For Review Only

# Acknowledgements

Firstly, I am truly grateful to my brother (Niranjan Agrawal) and my mother (Bharti Agrawal) for always being there for me. Leaving a decent job offer for the GATE preparation was an easy decision for me only because of their amazing support. They always tried their best to make sure that my mind state was calm and relaxed, especially during the initial phase of my research journey. They provided all the resources to ensure that my research was not affected when I was at home due to the pandemic. They believed in me, and their words motivated me never to give up. My brother's guidance and his efforts really helped me to do my research in a more structured way. He also directed me to the relevant persons who had expertise in my domain. Almost once every week, he discussed my project with me, which helped me speed up my work.

Also, I want to thank my advisors, Prof. Kanchi Gopinath and Prof. Vinod Ganapathy, for always believing in me; giving me sufficient time and space to learn new things. Working under them was a great experience, and I learned a lot from them. They were always there for me, like recommended good research papers to get started, gave excellent ideas to work upon, helped in my writing skills, provided the full financial assistance for my entire stay at IISc, and provided all the resources required for my research work. I am truly grateful to you for everything. I would also like to mention Prof. Uday (my initial faculty advisor), who pushed me to complete my RTP criteria in the 1st semester. Also, I would like to thank Prof. Govind, for his invaluable support during my tough time at IISc in the 1st semester.

A special thanks to my close friends - Soumya and Utkarsh, for making my stay at IISc memorable. All the things that we do - like, ordering food, cycling, walking, bounce tripling, random movie plans, coding discussions for placements, being there for each other in the one's tough times, and going for the meals together in D-Mess, really helped me to do my research with more enthusiasm. I would also like to mention my other close friend Rajat, who was always concerned about my research work and always tried his best to help me in the coursework projects and my research work. During the pandemic time, he recommended me to join CSL weekly reading group. Representing the papers here and the relevant discussions helped me

For Review Only

## Acknowledgements

understand a few concepts that were beneficial in my research project. I would also like to mention my friends - Dushyant, Raja, Dhrumil, Gaurav, Madhurima, Akshay, Chaitra, Vinayak, and Harsh, who were there on my IISc journey.

I am truly grateful for the CASL members - Ajinkya, Abhishek, Amrita, Ashish, Anup, Pratyush, Nilesh, Parth, Nikita as well as CSSL members- Kripa, Rounak, Chinmay, Nikita, Isha, Arun, Ajay, Rakesh, Akash, Subhendu, Eikansh, Gokul for being awesome labmates and for all the discussion related to my research work. Going out with these guys -celebrating someone's success, birthday celebration, etc. always refreshed my mind and allowed me to continue my research work with more enthusiasm.

I would also like to thank Mrs. Kushael and the other non-teaching staff of CSA for all the administrative work. They always try to help the students in the best possible manner. I would also like to thank other members of IISc who have taken great care of all the people in the campus community during the pandemic time (e.g., conducting a vaccination drive) so that students can focus entirely on their research work. I am grateful to the various cultural classes in Gymkhana, like Zumba (by Jagadesh), which really helped me to do my research with more enthusiasm.

I would also like to thank all my relatives, who were there in my IISc journey. Special thanks to my uncle (Dr. Narendra Agrawal) for taking great care of me during a few times, when my health does not sound good. I would also like to mention my uncle (R. Anbu) for all the regular conversations over the call since my childhood and the interesting ideas that he suggested for my thesis. I would also like to thank Sreesh, Neeraj, and other non-IISc friends and teachers who tried to help me in my research project.

For Review Only

# Abstract

This thesis concerns the robustness of security checks in financial mobile applications (or simply financial apps). The best practices recommended by OWASP for developing such apps demand that developers include several checks in these apps, such as detection of running on a rooted device, certificate checks, and so on. Ideally, these checks must be introduced in a sophisticated way, and must not be locatable through trivial static analysis, so that attackers cannot bypass them trivially. In this thesis, we conduct a large-scale study focused on financial apps on the Android platform and determine the robustness of these checks. Our study shows that among the apps with at least one security check, $> 50\%$ of such apps at least one check can be trivially bypassed. Some of such financial apps we considered have installation counts exceeding 100 million from Google Play. We believe that the results of our study can guide developers of these apps in inserting security checks in a more robust fashion.

For Review Only

# Contents



For Review Only

CONTENTS



For Review Only

# List of Figures



For Review Only

# List of Tables



For Review Only

# List of Algorithms



For Review Only

# List of Code Snippets



For Review Only

# Chapter 1

# Introduction

For Review Only

Mobile devices have become an integral part of the payment ecosystem. Payments are facilitated by financial applications (apps), which have in turn soared in popularity. In 2021, mobile finance apps reached 573.1 million downloads in the US, which is nearly an increase of 19% from 481.9 million in 2020 [27]. Another study analyzed the installation statistics of over 2,000 financial apps, reporting over 4.7 billion installations of these apps [33]. Digital wallets have also emerged as the second-most popular in-store payment method [35]. According to the latest report by the Reserve Bank of India (RBI), there has been an overall credit transfer amounting to 41.03 trillion rupees for the year 2020-21 using the Unified Payments Interface (UPI) [46]. UPI has emerged as the *de facto* mobile payment standard in India, and UPI-based transactions have increased by 70× over the last four years.

Given the increasing dependence on the financial app ecosystem and the sensitive nature of the data handled by financial apps (including the bank/card details of the payees and the payers), we set out to study a fundamental question: *Are financial apps vulnerable to the same security issues as other mobile apps?* We focused primarily on the Android ecosystem. We tried to understand the threats to mobile app security, and whether app developers follow the best practices to make apps more secure. Chief among the security concerns for mobile apps is code-tampering/repackaging, which leads to the insertion of malicious logic into the application. A prior study has shown that 5% to 13% of the apps on alternative marketplaces (i.e., sources other than Google Play) are repackaged [60]. In fact, code tampering was identified as one of the top 10 mobile app-related risks by the Open Web Application Security Project (OWASP) in 2016 [44]. Therefore, the OWASP guidelines recommend app developers equip apps with various self-defense mechanisms to defend against code tampering/repackaging. Although defenses such as SafetyNet [48] have been proposed and deployed widely, studies have shown that sometimes their use is improper [47], and the security of mobile apps continues to be reliant on the ability

of developers to incorporate self-defense mechanisms into their apps.

Previous work studied the various self-defense mechanisms in apps but has not been able to fill some gaps in this field. Kim et al. [58] bypassed root detection and app integrity checks of popular financial apps available in the Republic of Korea. However, their work is not entirely automated and did not consider the other self-defense checks recommended by OWASP. Berlato et al. [53] developed a static analysis tool, ATADetector, that detects anti-debugging and anti-tampering protection in apps, but did not bypass these protections. Ibrahim et al. [57] conducted an analysis on utilizing Google's SafetyNet API (used to perform device integrity and app integrity checks) and found that none of the analyzed apps invoking the SafetyNet API, uses it in an entirely correct way. This work manually bypassed the device integrity checks in 21 apps and did not reverse engineer enough to bypass the additional checks (related to device integrity) of other apps, since their focus was on SafetyNet. Zungur et al. [78] found that only 10.77% of apps from their dataset employ all the self-defense mechanisms recommended by OWASP. However, their study did not analyze the ease with which attackers could bypass these self-defense mechanisms assuming they are present in the app.

The Mobile App Security Verification Standards (MASVS), released by OWASP in 2018 [41], establish baseline security requirements for mobile apps. One of the recommendations for app developers given by MASVS is that all executable files and libraries belonging to an app must either be encrypted at the file level and that important code and data segments inside the executables must be encrypted or packed. Specifically, they recommend that trivial static analysis must not reveal important code or data.

In this thesis, we study the extent to which the developers of *financial apps* follow these recommendations and the robustness of the mechanisms in apps that incorporate them. We have analyzed 2,854 apps from the `FINANCE` category of Google Play [21]. To the best of our knowledge, this is the first work of its kind that has included such a large dataset targeting only financial apps. Our analysis is composed of two steps: First, we identify various security checks inside an app; and second, if such checks are present, we determine whether the checks are bypassable using a combination of static analysis and code instrumentation. Everything is automated, using open source tools. Recent work by Berlato et al. [53] revealed that app developers prefer to embed protections implemented within the Java to native code by a ratio of 99 to 1. Hence, our work focuses on protections inserted within the Java code of the application itself. We first conduct a keyword-based search of APIs used to insert self-defense checks in apps. If the checks can be located with a simple keyword-based search, an attacker can easily bypass them. We then attempt to verify whether the checks can indeed be bypassed by adding instrumentation to the app. This process, therefore, serves to determine the robustness of the

For Review Only

inserted self-defense checks in the apps.

Our study focuses on the four major kinds of self-defense checks that are inserted in apps: (1) anti-tampering, (2) emulator detection, (3) root detection, and (4) dynamic instrumentation framework detection. We describe these checks in more detail in the subsequent sections. For the rest of the thesis, we will use the term *self-defense check* (SDC) to refer to any of the checks enlisted above.

To determine whether a particular SDC is active in an app, we observe the differences between the execution flow of the original app and the compromised app (or the app running on a compromised device). That is, we run the original app and also run a rewritten version of the app and look for differences in their execution flow by analyzing the execution logs (using logcat [40]) and front-end UI (User Interface) elements (using UIAutomator [26]).

Our conclusions are that the SDCs inserted by a significant fraction of app developers are ineffective and easily bypassable. It is worrisome that many such apps have installation counts in the order of 10 million in Google Play. In particular, with respect to the SDCs we consider, we find that out of the 2,854 financial apps studied, anti-tampering, root detection, emulator detection, and dynamic instrumentation framework detections are present only in 45.8%, 47.7%, 46.5%, and 45.3% of the total apps, respectively. Moreover, we were able to successfully bypass the corresponding checks in 676, 378, 308, and 338 financial apps, respectively. Our work successfully bypassed all the SDCs that an app has in 584 instances of financial apps. We found that, of the apps in our dataset with at least one SDC, more than half have at least one trivially bypassable SDC.

## 1.1 Outline

The rest of the thesis is organized as follows:

- Chapter 2: We provide the background required to understand the thesis. We first discuss the basics of the android app, and then we describe the various SDCs that this work has focused on and the techniques through which developers can put these checks in their apps.

- Chapter 3: We first discuss the threat model, followed by how we collected a large number of apps from Google Play. Then, we discuss the techniques used to detect each of the SDCs in mobile apps and how we bypassed them.

- Chapter 4: We show the accuracy of our automated approach and the results obtained after analyzing 2,854 financial apps.



For Review Only

- Chapter 5: We discuss the characteristics of a few Indian financial apps analyzed in-depth.
- Chapter 6: We put up the recommendations for the app developers to make the attacker's job very tough to bypass such checks.
- Chapter 7: We discuss the limitations of our approach with respect to declaration of the SDC detection in apps and bypassing such checks.
- Chapter 8, 9: We mention the previous works related to this area, discussed its limitations, and finally conclude the thesis.
- Appendix A: We show the list of banks to which we have disclosed the problems.

For Review Only

# Chapter 2

# Background

For Review Only

Every Android app has a corresponding Android Package, which is an archive file with a `.apk` suffix. There are four different app components: Activities, Services, Broadcast receivers, and Content providers. While an *activity* represents a single screen with a UI, a *service* keeps the app running in the background. A *broadcast receiver* component enables the app to receive the system-related information, and a *content provider* manages a shared set of app data that one can store on any other persistent storage location that the app can access [32].

During the `apk` build process, the Java code of the app is compiled to Dalvik bytecode, and this bytecode is present in the *.dex file, which is part of the `apk`. Android also provides a feature where developers can write the code in C/C++. These C/C++ files are compiled to a shared object (*.so), and these *.so files are present in the `apk` under the lib directory.

Android mandates that every `apk` file must be digitally signed by the developer's public/private key pair [8]. Every time an app is being installed, Android checks whether the `apk` is digitally signed. If it is, Android verifies the signature of the `apk`, with the help of the developer's public key certificate, both of which are part of the `apk` itself. The app is not installed if the signature verification fails [51]. This section describes the various SDCs this work has focused on, as well as recommended by OWASP.

- **Anti-Tampering**: App developers include this check to protect apps against repackaging attacks. Repackaging of a mobile app is the modification of the existing app by inserting some code using a reverse engineering tool, then regenerating the `apk` and signing it with own 's public/private key (i.e., with the help of the Java KeyStore (JKS) [17] file). In case there is no anti-tampering protection in the app, an attacker can get the original app from Google Play, understand the semantics of the app by disassembling or decompiling the `apk`, insert the malicious code, and redistribute this app on unofficial marketplaces. For

example, we have produced repackaged versions of a few Indian financial apps that can store the end-user's sensitive details (like credit card number, bank profile login credentials) in our local server. Hence, it is essential to incorporate anti-tampering protection, and there are a few mechanisms through which developers can do so:

  - **Certificate Check**: Android provides APIs to retrieve the public key certificate details used to sign the `apk`. If the victim is using a modified version of the app, then the public key certificate used to sign the app will differ from the developer's public key certificate. Hence using such APIs, the app can determine if its integrity is compromised.
  - **Installer Verification**: This is an extra layer of protection that the developer can use to make sure that the user has indeed installed their app from the source (e.g., Google Play) where the developer has published it. Android offers APIs [16] that provide some insights to developers regarding the installation source of the app.

- **Root Detection**: Root detection checks are ways to ensure that the app is not running on a compromised device (i.e., on rooted device). If the device is rooted, a user with superuser privileges has access to modify some directories which is not possible on a non-rooted device. For example, in our work, we were able to run the unsigned version of many apps on the rooted device by making relevant changes in the directory where the `apk` resides. These checks are related to finding the presence of the superuser (`su`) binary, malicious apps (that helps in rooting) on the device, and checking whether any system-related directories are writable.

- **Emulator Detection**: This SDC checks whether the app runs on a physical device or atop a mobile device emulator. Google's SafetyNet [48] can be used to perform this check. Another alternative is to get the app's device-related information (like Brand, Manufacturer, Model, DeviceId, etc.) using the Android APIs [13],[25]. Comparing these values against the value of a real physical device is, thus, possible.

- **Dynamic Instrumentation Framework Detection**: These frameworks can alter the flow of execution of the app without statically modifying the `apk`. Many such frameworks, (e.g., Frida [14], Xposed Framework [52], etc.) are used in the Android ecosystem. App developers should put this SDC to ensure that such a dynamic instrumentation framework is not running on the device when their app is running.

For Review Only

# Chapter 3

# Methodology

In this chapter, we first discuss the threat model from the user's side regarding the source of app installation and the assumptions made for an attacker, followed by the technique for downloading a large number of apps from Google Play in an automated way. Then, we discuss the methodology used to find the SDCs in the apps. Later, we discuss the techniques to bypass these checks in the apps by performing code instrumentation. At last, we discuss the system configurations in which these apps were launched to determine the SDC.

## 3.1 Threat Model

To conduct our analysis, we assume that the end-user has given permission to Android for installing the apps from the sources like WhatsApp, Google Drive, File Transfer Apps, etc. This assumption is required because, by default, Android does not allow the users to install the apps from sources other than Google Play. However, we feel that this assumption is reasonable in many developing countries where alternative app stores are very popular. For example, a survey shows that 30% of the total app installations in a country like India occur through a file transfer app [23].

We assume that an adversary does not have raw access to the app's source code (which would reveal the location of SDCs) and the developer's private key. However, we assume that the adversary has expertise in various static reverse engineering tools (like apktool [7], Ghidra [37], radare2 [45], JADX [38], etc.) to decipher the semantics of the app for inserting the malicious code that can bypass the SDCs. Adversaries typically use these methods to produce tampered apps, then upload them to the alternative marketplaces. Also, prior studies have shown that 5% to 13% of all the apps in the alternative app marketplaces are repackaged [76]. We also assume that an adversary can compromise the end-user's Android device by rooting.



For Review Only

Following that step, the attacker can alter the execution flow of the running app and perform various malicious activities like modifying the network traffic, etc.

## 3.2 Dataset Collection

Every Android app is uniquely identified by its package name. Currently, there is no mechanism to obtain all the package names under the `FINANCE` category of Google Play. At first, we used the "Google Play Scraper" [15] to obtain the package names of around 2,300 apps using keywords such as UPI (Unified Payment Interface, is an instant mobile payment feature that powers multiple bank accounts into a single mobile application, seamless fund routing & merchant payments into one hood [42]), Banking Apps, Gramin ("rural") bank apps, Small Finance Banking, Regional Rural Bank, etc., and also through the assigned developer ID used to publish the app on Google Play. Later, we found that this program does not accurately yield a complete list of all the package names when queried with a given developer ID. With the help of a front-end web automation tool (like Selenium [22]), we crawled the web page for the given app's package name to find whether it belongs to the `FINANCE` category and retrieved the developer ID for that app. Subsequently, we crawled Google Play website for the given developer ID to get more apps. Using this methodology, we obtained package names of 3,371 financial apps. However, we found that some of the apps are unavailable in our country, a few are premium apps, and a few are no longer available in Google Play. Table 3.1 categorizes the unavailable apps for various reasons.

| Apps removed from Google Play | Premium Apps | Apps unavailable for India |
|---|---|---|
| 301 | 92 | 124 |

Table 3.1: Reason-wise count of the excluded financial apps

Our final dataset count is 2,854 financial apps. We downloaded all of these apps from Google Play in an automated way using Appium [9] and UIAutomator viewer [28] (which are front-end automation tools used in the software industry for UI Testing). After the app installation, we fetched the `apk` using Android Debug Bridge (`adb`) [1]; thus, we conducted our analysis on the latest version of the apps.

## 3.3 SDC Detection

This section discusses the methodology to find various SDCs in the app. The high-level idea of our methodology is to observe the execution flow of the original app and the compromised



For Review Only

app (or the app running on a compromised device like rooted device, emulator, etc.). If it is different, we declare that a SDC is detected in the app. To observe the execution flow, we have analyzed the UI elements of the app's activity and logs for each run after it was launched for a minute. Suppose either different UI elements or logs (after relevant filtering) are observed in these two launches. We then declare that the execution flow is different.

The SDC related code should get executed in the app's lifetime, and we believe that it should get executed immediately after the app's launch. If it is not the case, it will be straightforward for the attackers to produce a tampered version of the app in such a way that can steal the sensitive information about the users (e.g., contact details, getting the device information, login credentials, etc.) by injecting the malicious code at the appropriate location.

For retrieving UI elements, we first capture the complete hierarchical structure (i.e., the view hierarchy) of the app's current activity in `xml` format using UIAutomator [26], a tool present in the mobile device. It contains various nodes, and each node represents a UI element in the app's activity. (We showed one such node that presents a UI element "customer id" of type EditText, in Listing 3.1.) Then from each node, we retrieved the value of the attribute `resource-id`. If the set of such values fetched differs in the two runs, we conclude that different UI elements are observed. Zungur et al. [78] have also used the tool UIAutomator to analyze the app's UI to determine the SDC detection in the app. Apart from analyzing the UI, we have also done the log analysis, as sometimes, the UIAutomator fails to capture the app's UI elements.

For the log analysis part, we captured the logs using Logcat [40]. We fetched the logs after allowing the app to run for a minute, using the process ID of the app, and by searching other log statements that have the package name of the app using the `grep` command. We cleared the log buffer before the launch of the app to ensure that we get the logs for the current run only.

We did the log analysis to identify the difference in the following three behaviors when the compromised app was launched (or the app launched on the compromised device) with respect to the original app launched on the non-rooted real device:

- Various activities displayed in the app's launch.
- App exits after leaving some message to an end-user.
- App crashes immediately after its launch on the device.

So, once the logs were captured, we focused on the log messages from the `ActivityTaskManager` [4] and `ActivityManager` [3] to extract the activity names displayed. If different activities are



For Review Only

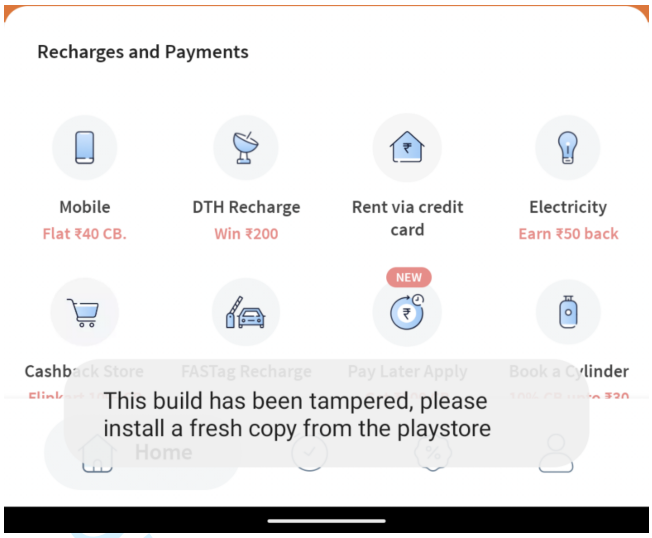



Figure 3.1: `com.freecharge.android` exits after showing the toast message in its repackaged version.

observed in these two launches, there is a different flow of execution. If not, we fetched the log messages from `NotificationManagerService`[1] [20] to identify whether an app showed an error message to an end-user (e.g. in the form of a Toast message [49], shown in Figure 3.1) when being launched in a hostile environment. If that is also not the case, we find whether the app exits immediately after being launched by retrieving its process ID after the launch for a minute using dumpsys [36] (a built-in tool, which is present in the mobile device). This tool provides the process ID of running apps only. So, if the process ID is not retrieved, it means the app is no longer running on the device. Hence, if any of the these three behaviors is observed from the logs, we declare that the execution flow is different.

So, if the execution flow is identical in the two runs (i.e., no difference found from the UI analysis and log analysis), it means SDC is not detected in the app; otherwise, we declare SDC detection in the app. This is shown in Algorithm 1.

```
<node index=''0'' text='''' resource-id="customerId" class=''android.widget.
    EditText'' package=''com.suryodaybank.mobilebanking'' content-desc=''''
    checkable=''false'' checked=''false'' clickable=''true'' enabled=''true''
    focusable=''true'' focused=''false'' scrollable=''false'' long-clickable=''
    false'' password=''false'' selected=''false'' bounds=''[99,427][984,529]'' />
```

Listing 3.1: Partial View hierarchy for Figure 3.2a in `xml` Format.

[1] W/NotificationService(1797): Toast already killed. pkg=`com.freecharge.android` token=android.os.BinderProxy@29fd2cf



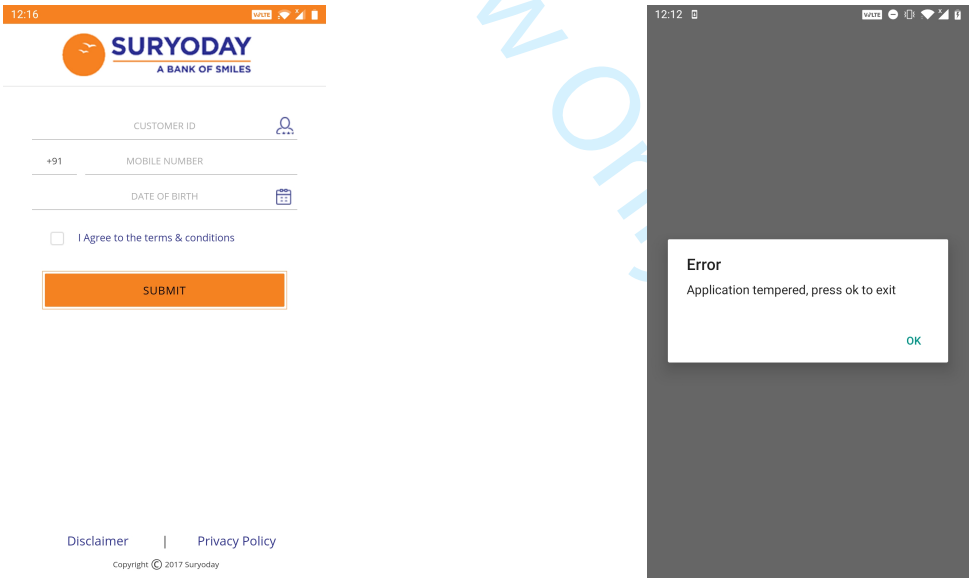
https://mc04.manuscriptcentral.com/iisc_mtech

**Algorithm 1** SDC Detection in Android apps.

**Input:** Original `apk`
**Output:** A boolean value indicating whether SDC is present in the app.

1: Capture the UI elements and logs for the original app launched in non-rooted real device and compromised device.
2: $UI_{orig} \leftarrow$ App's UI elements on non-rooted real device.
3: $UI_{comp} \leftarrow$ App's UI elements on compromised device.
4: $Logs_{orig} \leftarrow$ App's logs on non-rooted real device.
5: $Logs_{comp} \leftarrow$ App's logs on the compromised device.
6: **if** $UI_{orig} \neq UI_{comp}$ **OR** $Logs_{orig} \neq Logs_{comp}$ **then**
7: SDC is detected.
8: **else**
9: SDC is not detected.
10: **end if**

For Review Only




(a) Original app's UI (b) Tampered app's UI

Figure 3.2: Different UI for the app `com.suryodaybank.mobilebanking` when original and its repackaged version is launched

```
<node index=''0'' text="Application tempered, press ok to exit"
    resource-id="android:id/message" class=''android.widget.TextView'' package=''
    com.suryodaybank.mobilebanking'' content-desc='''' checkable=''false'' checked
    =''false'' clickable=''false'' enabled=''true'' focusable=''false'' focused=''
    false'' scrollable=''false'' long-clickable=''false'' password=''false''
    selected=''false'' bounds=''[72,1137][1008,1196]''/>
```

Listing 3.2: Partial View hierarchy for Figure 3.2b in `xml` Format.

Figure 3.2 depicts this method for detecting an SDC by analyzing the app's UI. Their corresponding partial view hierarchy is shown in Listings 3.1 and 3.2.

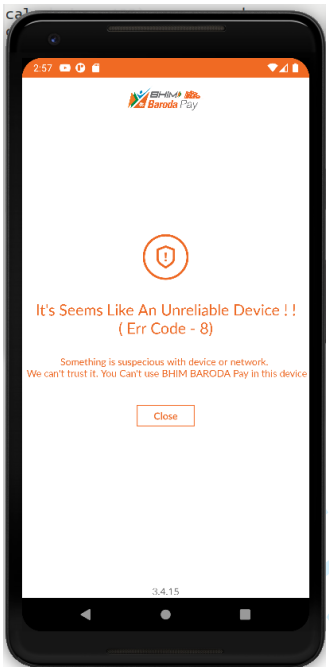
For the baseline setup, we first launched the original app on the non-rooted real device for a minute using `adb`. Following that, we captured the view hierarchy of the app's current activity (dumped in `xml`) and the logs related to this run. Before launching the app, we found the runtime permissions that an app may request from the user (using AAPT2 [31]) and provided the same using package manager (`pm`) tool through `adb` to suppress any permission-related dialog.

For the anti-tampering check, we produced the repackaged version of the app by resigning the `apk` through apksigner [6] by using our Java KeyStore (JKS) [17] file, which is a secure file format used to hold various information like public key, private key, certificate information. (Some apps are packaged with more than one `apk` – base `apk` and one or more "split" `apk` [30]. We have handled such scenarios and re-signed the split `apk` too.) Upon producing the app's repackaged version, we install the app using `adb`, grant all the permissions, launch it for the same duration of time on the non-rooted real device, instead of a compromised device. Following that step, we capture the app's UI-related information and the logs related to this run. If there is a difference in the UI elements or from the log analysis with respect to the run of the original app, we declare that the anti-tampering check is present.

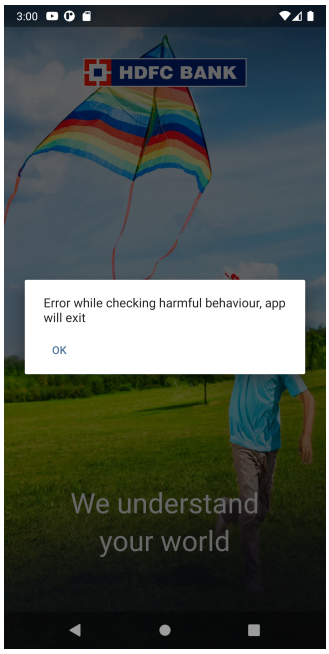
We have used the same technique to find the three other SDCs in the app, but the difference is that we did not produce a repackaged app. For root detection and emulator detection checks, original app is run on the rooted device and emulator, respectively. For dynamic instrumentation framework detection, we run the original apps on the device in which such a framework (e.g., Frida, Xposed Framework) is running. Figures 3.3 and 3.4 show the way app reacts when it runs on a compromised device. All the app installations across various devices, launching the app, capturing the logs, and the UI elements - each step is automated through `adb`.

We did not use static analysis for SDC detection as it cannot obtain accurate results if the related code is not included in the `apk`, but it is in the code loaded dynamically at the runtime

For Review Only

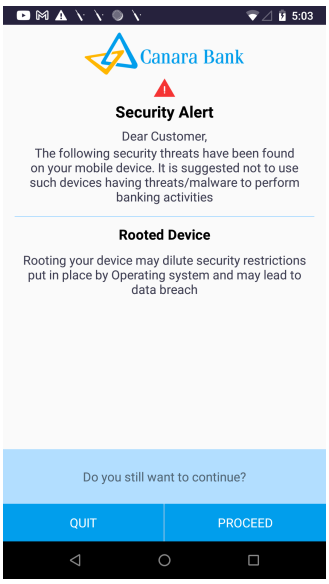



(a) `com.bankofbaroda.upi`

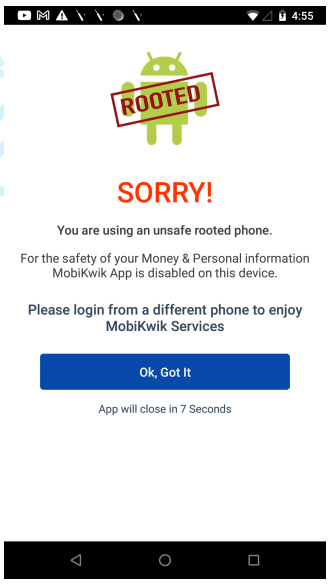



(b) `com.snapwork.hdfc`

Figure 3.3: Error messages shown by the app when it is run on an emulator.




(a) `com.canarabank.mobility`




(b) `com.mobikwik_new`

Figure 3.4: Error messages shown by the app when it is run on a rooted device



https://mc04.manuscriptcentral.com/iisc_mtech

(e.g., from the backend-server or the code is initially encrypted and gets decrypted at runtime). It will also give false results if the code related to the given SDC is present but not executed at runtime. Pham et al. [67] found that sensitive Android APIs were not locatable through static analysis for many apps but found that these APIs are being invoked when the app was launched. A possible reason they mentioned is that such requests were dynamically loaded at runtime.

## 3.4 Bypassing SDCs

After finding SDCs in the apps, the next step is to determine whether the checks can be bypassed in those apps. This section discusses the approach used to bypass each SDC and the generation of instrumented apps. Once the instrumented apps are produced, we launch them on the same device where the untampered app is launched for the detection purpose (e.g., emulator for emulator detection, rooted device for root detection, etc.). We capture the view hierarchy of the app's current activity (dumped as an `xml`) and the logs for this run. We compare this dump and the logs with the original app's run in the non-rooted real device with the same technique that we used for the SDC detection. If the execution flow is the same, our work concludes that we have successfully bypassed the check. This is shown in Algorithm 2.

**Algorithm 2** SDC Bypass Detection in Android apps.

**Input:** Original `apk` (with SDC) and its modified version.
**Output:** A boolean value for whether SDC is bypassed.

Launch these apps for a minute and capture the UI elements and logs for the original app and the modified app.
$UI_{orig} \leftarrow$ UI elements of the original app.
$UI_{modif} \leftarrow$ UI elements of the modified app.
$Logs_{orig} \leftarrow$ Logs of the original app.
$Logs_{modif} \leftarrow$ Logs of the modified app.
**if** $UI_{orig} \neq UI_{modif}$ **OR** $Logs_{orig} \neq Logs_{modif}$ **then**
    SDC is not bypassed.
**else**
    SDC is bypassed.
**end if**

For code instrumentation, we used the reverse engineering tool Apktool [7]. For each of the classes bundled in the dex file (part of the `apk`), there is a corresponding `.smali` file generated after disassembling the `apk`. These `smali` files are human-readable, and we therefore instrument the `smali` code as an indirect means to modify the Java code and then regenerated the `apk` using the same tool and signed it using apksigner [6]. This entire process of doing the code



For Review Only

instrumentation and producing the modified `apk` is completely automated. For each SDCs, we find the relevant APIs for the implementations of self-defense checks from the resources like Android Developers [5], OWASP's MASVS [41]. After that, we located the `smali` files in which these APIs are invoked, using the `grep` command and then, we made the required modifications in these `smali` files. Now, we discuss the methodology used to produce the instrumented apps to bypass each SDC.

- **Anti-Tampering:** Certificate check and installer verification are the two ways to have anti-tampering protection in an app. First, we discuss the methodology adopted to bypass certificate checks.

  We retrieved the public key certificate used by the developer to sign the `apk`. We resorted to different strategies based on the developer's method of signing the `apk`. If it is signed using the v1 signing scheme [50] (Android supports three application signing schemes for signing the `apk`: v1 signing scheme is one such technique, based on JAR signing [39]), we first locate the *.RSA file in the META-INF directory of the `apk`. This *.RSA file contains the information related to the developer's public key, certificate details, information about the subject and the issuer of the certificates, etc. We obtained the certificate details using this file with the help of `openssl` [43].

  However, app developer need not use the v1 signing scheme; there are approximately 100 such apps in our dataset. To address this scenario, we installed all such apps on our mobile device and developed an app that retrieves the certificate details of these apps.

  Once the certificate details are retrieved, for each of the methods defined in the `Signature` [24] class, we locate the `smali` files in which these methods are invoked to perform the certificate validation. Just before the instructions, where such methods are invoked, we injected a `smali` code segment that will create a new instance of `Signature` class with the parameter to its constructor being the developer's public key certificate (fetched earlier) in hexadecimal-string format. For this, we first find the unused virtual register to store this string. Then, we store this `Signature` instance in the virtual register (`vI`, where I $\in \mathbb{W}$), which is supposed to hold the app's certificate details fetched during the runtime. Now, wherever the methods of the `Signature` class are invoked for the validation, the return value of these methods will be related to the developer's certificate details.

```
1 invoke-virtual {v0, v1, v2}, Landroid/content/pm/PackageManager;->
      getPackageInfo(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;
2 move-result-object v0
```



https://mc04.manuscriptcentral.com/iisc_mtech

For Review Only

```
3  iget-object v0, v0, Landroid/content/pm/PackageInfo;->signatures:[
       Landroid/content/pm/Signature;
4  const/4 v1, 0x0
5  aget-object v0, v0, v1
6  invoke-virtual {v0}, Landroid/content/pm/Signature;->toCharsString()Ljava
       /lang/String;
7  move-result-object v1
8   const-string v2, ``Expected Certificate Details"
9  invoke-virtual {v1, v2}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
10 move-result v3
```

Listing 3.3: Anti-tampering check `smali` code snippet.

We discuss the `smali` code snippet line by line for the anti-tampering protection, shown in Listing 3.3. At a high level, this code snippet fetches the certificate details of the app and compares it with the expected one. In line 1, `getPackageInfo()` method gets invoked, which is defined in `PackageManager` class in `android.content.pm` package. This method takes two parameters - package name (stored in virtual register `v1`) and a flag value to fetch the desired properties of the app (stored in virtual register `v2`). Its return value is a `PackageInfo` object (stored in virtual register `v0`, shown in line 2). One of its data members is an array of `Signature` objects, accessed in line 3. This array stores the app's certificate details used while signing the `apk`. The instructions at lines 4 and 5 access the first element of this array and store it in the virtual register `v0`. Now, the certificate validation is performed through the methods of the `Signature` class. As shown in line 6, one such method used is `toCharsString`(), and its return value is stored in the register `v1` shown in line 7. Virtual register `v2` holds the expected certificate details (in the string format) shown in line 8. The instruction in 9 checks whether the public key certificate fetched is the same as the expected value, and this result is stored in the virtual register `v3` shown in line 10.

```
1  invoke-virtual {v0, v1, v2}, Landroid/content/pm/PackageManager;->
       getPackageInfo(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;
2  move-result-object v0
3  iget-object v0, v0, Landroid/content/pm/PackageInfo;->signatures:[
       Landroid/content/pm/Signature;
4  const/4 v1, 0x0
5  aget-object v0, v0, v1
6  const-string v6, "Developer's Public key Certificate Details in hexa-decimal"
7  invoke-direct {v0, v6}, Landroid/content/pm/Signature;-><init>(Ljava/lang/String;)V
```



https://mc04.manuscriptcentral.com/iisc_mtech

For Review Only

```
8  invoke-virtual {v0}, Landroid/content/pm/Signature;->toCharsString()Ljava
       /lang/String;
9  move-result-object v1
10 const-string v2, ``Expected Certificate Details"
11 invoke-virtual {v1, v2}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
12 move-result v3
```

Listing 3.4: Anti-tampering check bypass `smali` code snippet.

Listing 3.4 shows how we bypass this check. Two new instructions have been added before invoking one of the methods of `Signature` class. Line 6 makes the virtual register `v6` hold the reference to a string, which is the developer's public key certificate in hexadecimal. The next instruction in line 7 creates a new instance of the `Signature` class with the developer's public key certificate passed to its constructor. In instruction 8, virtual register `v0` will store the developer's certificate information; hence we can bypass the check even though the installed `apk` is signed using a different certificate.

This code instrumentation technique can also bypass the check for the scenarios when the certificate validation happens on the server side as the code related to fetching the certificate details is part of the `apk` and can be located by a keyword-based search.

Next, we discuss the technique to bypass installer verification. We first find the Android APIs used to fetch the source of the app installation. One such API is `getInstallSourceInfo()` which takes package name as a parameter. After that, we find the register which is passed as a parameter to this API. We overwrite this register's value with `com.android.vending`, which is the package name of Google Play. (An attacker can also overwrite the register value with the package name of any app which has been installed from Google Play on the mobile device.) Now, whenever this API is invoked regarding the app's installation source, the result will be Google Play as the virtual register now holds the package name of Google Play. We can thus bypass the installer verification check successfully in this manner.

An app can also have a dependency on third-party libraries to provide some functionality. We excluded these third-party libraries' equivalent smali files to perform the code instrumentation as these libraries can have the APIs used to perform the anti-tampering check, but for different purposes. We relied on ATADetector tool [53] to obtain the list of such third-party libraries.

- **Emulator Detection**: The high-level idea to bypass the emulator detection check is by rewriting the register value that stores the build-related information (like Brand, Man-



For Review Only

ufacturer, Model, DeviceId etc.) of the app's device with the build information of a non-rooted real device.

To achieve this goal, we went through the source code of the `Build` class[12] in `android.os` package, to obtain the information about the various data members, methods that can be used to fetch the build-related details about the app's device. Next, we developed an app that runs on the physical device and give the values of these data members and methods. Now, we locate such data members, methods in the `smali` files and overwrite the equivalent register's value with the corresponding value fetched earlier. Listings 3.5 and 3.6 show the `smali` code snippet of the emulator detection check and how we bypassed it, respectively.

```
1 sget-object v2, Landroid/os/Build;->FINGERPRINT:Ljava/lang/String;
2 const-string v3, ``generic''
3 invoke-virtual {v2, v3}, Ljava/lang/String;->startsWith(Ljava/lang/String
    ;)Z
```

Listing 3.5: Emulator detection check `smali` code snippet.

```
1 sget-object v2, Landroid/os/Build;->FINGERPRINT:Ljava/lang/String;
2 const-string v2, "google /sunfish /sunfish:11 /RQ3A.210605.005/7349499:user
    /release-keys"
3 const-string v3, ``generic''
4 invoke-virtual {v2, v3}, Ljava/lang/String;->startsWith(Ljava/lang/String
    ;)Z
```

Listing 3.6: Emulator detection check bypass `smali` code snippet.

In Listing 3.5, line 1 fetches the build-related information of the app's device with the help of `FINGERPRINT`, a static data member of the `Build` class, and the return value is stored in virtual register `v2`. Line 2 makes the virtual register `v3` store the reference to a string "generic". The instruction in line 3 checks whether the device's fingerprint starts with "generic" using the method `startsWith()` defined in `String` class. Listing 3.6 shows the code snippet in which we have overwritten the register's value (which stores the build information of the app's device) with that of the real device (fetched earlier). The injected code is shown in line 2 of Listing 3.6.

- **Root Detection**: To bypass the root detection check, we first prepared a list of malicious app package names used for rooting the device and a list of directories that an app checks either for the presence of the `su` binary or for the read-write permissions. We found that if an app has this SDC, then directory names, the package name of these malicious apps,



https://mc04.manuscriptcentral.com/iisc_mtech

For Review Only

can be found in the `smali` code as a string. So, we located the `smali` files where we have such names as a string using the `grep` command, and then we overwrote it with a different name. Hence, even if the malicious app or the `su` binary is present, it will not get detected as we modified the virtual register's value holding such strings to a different name. Listings 3.7 and 3.8 shows the code related to the root detection check and how we bypassed it, respectively.

```
1 const-string v3, ``/system/xbin/which"
2 const-string v4, ``su"
3 filled-new-array {v3, v4}, [Ljava/lang/String;
4 move-result-object v3
5 invoke-virtual {v2, v3}, Ljava/lang/Runtime;->exec([Ljava/lang/String;)
    Ljava/lang/Process;
```

Listing 3.7: Root detection check `smali` code snippet snippet.

```
1 const-string v3, ``/system/xbin/which"
2 const-string v4, "Someword"
3 filled-new-array {v3, v4}, [Ljava/lang/String;
4 move-result-object v3
5 invoke-virtual {v2, v3}, Ljava/lang/Runtime;->exec([Ljava/lang/String;)
    Ljava/lang/Process;
```

Listing 3.8: Root detection check bypass `smali` code snippet.

Listing 3.7 shows a `smali` code snippet to find the location of `su` binary in the device using `which` command. The first two instructions at line 1 and 2 make the virtual register point to the corresponding string used for executing this command. The instructions at lines 3 and 4 create an array with these strings, and virtual register `v3` points to this array. The last instruction executes the command `"/system/xbin/which su"` to find the location of the `su` binary using the `exec()` method of the `Runtime` class. Its return value is of type `Process` class, which can be further used to find the location of `su` binary, if it exists. Listing 3.8 shows how we bypass this check by renaming the string `su` with another word, shown at line 2. When the last instruction gets executed, the output will reveal the path of the "Someword" binary, which does not exist. So, in this way, we have bypassed this check.

- **Dynamic Instrumentation Framework Detection**: We found that if an app has this SDC, the app will query the presence of such frameworks in the mobile device by checking whether such a framework (e.g., Frida) is running or by checking whether the related app



https://mc04.manuscriptcentral.com/iisc_mtech

For Review Only

is installed (e.g., `de.robv.android.xposed.installer` [52]). We found that the related keywords are passed to the desired methods as a string. Similar to the methodology used in the case of root detection, we also bypassed this check by overwriting such strings with a different word. So, now even if a hooking framework like Frida runs, it will not be detected.

If an app has multiple SDCs, then to detect whether we have bypassed a particular SDC, sometimes, we must bypass the other checks also. For instance, in the case of an app with both anti-tampering and root detection protections, to identify whether our work has bypassed the root detection check, it has to bypass the anti-tampering check too because modifying the code to bypass the root detection check will compromise the integrity of the app. Thus, the instrumented app must be produced in such a way that it can bypass both root-detection and anti-tampering protections. Hence, failure to bypass the anti-tampering protection does not imply that the root-detection check has been bypassed because the UIs of the app for the two runs would still be different. Therefore, in our work, we first identified the various SDCs for each of the apps and, accordingly, generated instrumented versions of these apps.

## 3.5 System configuration

For the baseline setup, we installed and launched all the original apps on Google Pixel 4a with 6 GB LPDDR4* memory and a 64-bit octa-core Qualcomm r5 Snapdragon™ 730G processor. For the rooted environment, we used LG Nexus 5X with 2GB LPDDR3 memory and a 64-bit hexa-core Qualcomm Snapdragon™ 808 processor. We used the Magisk app [19] to provide privileged root access to this device. For the emulated environment, we selected an emulator with `x86` compatible Android 11 (Google APIs) system image. They support ARM by default and provide dramatically improved performance when compared with full ARM emulation [34]. This emulator is run on a single computer with a quad-core Intel® Core™ i5-10210U, 8 GB of RAM, and an NVIDIA GeForce MX 250 GPU. For the dynamic instrumentation framework detection, we had two options to create an environment in which Frida could be launched. This can be done either by repackaging the app and including the Frida-related native library or running Frida on the rooted device. Since the former method compromises the app's integrity, we decided against that method and chose instead to create an environment to run such a dynamic instrumentation framework. The limitation of this strategy is that although an app may not have protection against Frida, it can have that against other dynamic instrumentation frameworks like Xposed Framework [52]. Zungur et al. [78] have also considered only Frida to identify this check. So, we ran Frida on the rooted device and then launched all the unmodified

For Review Only

apps here to determine this SDC in the app. We have written the code for the complete automation in Java, and we have stored all our results using the `MySQL`.

For Review Only

# Chapter 4

# Experimental Results

In this chapter, we first performed the calibration testing to know the accuracy of our automated approach. After that, we discuss the results of 2,854 Financial Apps.

## 4.1 Calibration of Automated Approach

In this section, we discuss the results obtained by analyzing 100 Indian financial apps. We used these results to calibrate the accuracy of our automated approach, which we then used for the large-scale analysis. For the analysis of 100 Indian financial apps, we selected apps from various bank categories: public sector banks (i.e., government-backed), private sector banks, regional rural banks, and small finance banks.

We manually ran these 100 apps on the compromised device. We determined the presence of a given SDC by observing whether the app showed an error message or abnormal behavior (e.g., an app crashing, etc.) after the app's launch. To find the anti-tampering protection, we ran the repackaged version of the app on the actual device and followed the same methodology. Table 4.1 shows the manual results of 100 Indian financial apps. The 2nd column gives the number of apps that have the corresponding check (out of 100 apps) and, from these apps, the third column mentions the number of apps in which we could bypass these checks successfully. We obtained the count for the last column by launching these instrumented apps on the same device (used for the SDC detection) and checking whether the app showed any error message. If it did not, it implied that we had successfully bypassed the check. Few banking apps for which we have the login credentials, we analyzed them end-to-end regarding SDC detection and bypassing it. For other banking apps, we tried to analyze till the point the app is not asking for any personal details.

Table 4.2 summarizes the results obtained through complete automation. Following the



For Review Only

| Self-defense check | Check Present | Check Bypassed |
|---|---|---|
| Anti-Tampering | 55 | 36 |
| Root Detection | 61 | 40 |
| Emulator Detection | 68 | 38 |
| Dynamic Instrumentation Framework Detection | 62 | 40 |

Table 4.1: Manual Analysis of 100 Indian financial Apps.

| Self-defense check | Check Present | False Positive | False Negative | Check Bypassed | Accuracy for check bypassed |
|---|---|---|---|---|---|
| Anti-Tampering | 41 | 1 | 15 | 32 | 87.5% |
| Root Detection | 66 | 3 | 9 | 35 | 82.9% |
| Emulator Detection | 54 | 1 | 8 | 39 | 82.1% |
| Dynamic Instrumentation Framework Detection | 58 | 4 | 15 | 28 | 82.1% |

Table 4.2: Automation results for 100 Indian financial Apps.

For Review Only

above-mentioned methodology of finding the presence of a check and bypassing it, we analyzed the same 100 apps, but this time through automation. Against each row of these checks we have mentioned the number of apps with these checks, the values corresponding to false analysis (both false positives and false negatives) with respect to the detection of the check, and bypassing such checks. The 5th column lists the number of apps, corresponding to each kind of check that our work has successfully bypassed through automation (we did not verify whether our work has actually bypassed the check), and the last column provides the accuracy statistics, i.e., the checks of how many apps have really been bypassed through our automation.

For example, the results from our automation show that it has bypassed the anti-tampering check in 32 apps. But compared against the list of apps for which our work has actually bypassed the check, we found that the anti-tampering check was successfully bypassed in 28 out of 32 apps. Similarly, we found three false positives for the root detection check, which implies that our automated approach incorrectly detected that the root detection check is present for these three apps. We also noticed eight false negatives for the emulator detection check, which means that our automated approach incorrectly detected that the emulator detection check is not there in these apps. The overall accuracy of our automated approach for SDC detection is 86% and accuracy for bypassing SDCs is 83.6%. The reasons for the false positives, false negatives and the accuracy with respect to bypassing these checks is discussed in the Limitations chapter 7.

## 4.2 Large-scale Results

After obtaining the results for 100 apps so as to calibrate, we have analyzed the entire dataset. We present the results of 2,854 financial apps. Through our study, we have addressed the following research questions:

- **RQ1:** How many apps have a particular SDC, and which SDC is the most popular among financial app developers?
- **RQ2:** What percentage of apps have at least one SDC?
- **RQ3:** How many apps have all the SDCs, and how many apps have none of these SDCs?
- **RQ4:** How many apps have a given SDC bypassable, and which SDC is the most trivial to bypass?
- **RQ5:** How many apps have all SDCs bypassable that it has and how many apps have SDCs, none of which are bypassable?

The results are very alarming. Table 4.3 shows the number of apps with a corresponding SDC and Table 4.4 provides the number of apps with at least one check and the apps we have bypassed the related checks. On the other hand, Table 4.5 summarizes the various combinations of SDCs in the dataset, and Table 4.6 summarizes the multiple combinations of the SDCs that have been bypassed in the apps with at least one check. E.g., an entry in the 4th row, 5th column in Table 4.5 shows that 42 financial apps have anti-tampering, root detection, and emulator detection checks. Similarly, an entry in the last row, last column in Table 4.6, shows that our automation has bypassed all the 4 SDCs in 37 financial apps.

| Apps | Anti-Tampering | Emulator Detection | Root Detection | Dynamic Instrumentation Framework Detection |
|---|---|---|---|---|
| 2854 | 1308 (45.8%) | 1326 (46.4%) | 1361 (47.6%) | 1292 (45.2%) |

Table 4.3: Large-scale analysis of financial apps for SDCs.

| Apps with at least one SDC | Anti-Tampering | Emulator Detection | Root Detection | Dynamic Instrumentation Framework Detection |
|---|---|---|---|---|
| 2081 | 676 | 308 | 378 | 338 |

Table 4.4: SDCs Bypassed Automation Results.

For Review Only

| **SDCs** | $\overline{RD}$ ,$\overline{ED}$ | $\overline{RD}$, ED | RD, $\overline{ED}$ | RD, ED |
|---|---|---|---|---|
| $\overline{AT}$,$\overline{DI}$ | 774 | 108 | 104 | 50 |
| $\overline{AT}$,DI | 64 | 42 | 106 | 316 |
| AT,$\overline{DI}$ | 346 | 75 | 63 | 42 |
| AT,DI | 54 | 49 | 35 | 644 |

**AT**:Anti-Tampering
**RD**:Root Detection
**ED**:Emulator Detection
**DI**:Dynamic Instrumentation Framework Detection
$\overline{\mathbf{X}}$:X SDC is not present.

Table 4.5: Various SDCs combinations in apps.

| **SDCs** | $\overline{RD}$ ,$\overline{ED}$ | $\overline{RD}$, ED | RD, $\overline{ED}$ | RD, ED |
|---|---|---|---|---|
| $\overline{AT}$,$\overline{DI}$ | 980 | 92 | 78 | 20 |
| $\overline{AT}$,DI | 78 | 23 | 47 | 87 |
| AT,$\overline{DI}$ | 485 | 25 | 47 | 16 |
| AT,DI | 12 | 8 | 46 | 37 |

**AT**:Anti-Tampering
**RD**:Root Detection
**ED**:Emulator Detection
**DI**:Dynamic Instrumentation Framework Detection
$\overline{\mathbf{X}}$:X SDC is not bypassed.

Table 4.6: Various SDCs bypass combinations in apps.

We found that any SDC is present in only $< 50\%$ of the apps. Table 4.3 shows anti-tampering, emulator detection, root detection, and dynamic instrumentation framework detection checks are present in 45.8%, 46.4%, 47.6%, 45.2% of the financial apps, respectively. From this, Root detection check is the most popular among app developers, which answers **RQ1**. As per Table 4.4, 2081 apps (i.e., 72.9%) have at least one SDC, that answers **RQ2**. Although a prior study [78] has addressed **RQ3**, its dataset comprises only a few financial apps compared to ours. As illustrated in Table 4.5, we found that 644 apps have all the SDCs. Among these, 17.1% of the apps have an installation count of more than a million in Google Play. We also found 774 apps have none of the SDCs, and among these apps, 113 financial apps have an installation count of more than one million. To answer **RQ4**, anti-tampering, emulator detection, root detection, and dynamic instrumentation framework detection checks are bypassed in 676, 308, 378, and 338 financial apps, respectively. Among the apps with at least one SDC, 1101 apps' (i.e., >50%) at least one SDC has been trivially bypassed. Among these apps, 37 financial apps have an installation count of more than 10 million. Our work bypassed the anti-tampering



For Review Only

check the most. As per Table 4.6, we found that out of the 2,081 apps with at least one SDC, none of the SDCs were bypassed in 980 apps. Our work bypassed all the SDCs that an app has in 584 financial apps ($\approx$ 28 %) which answers **RQ5**. Among these 584 apps, 84 financial apps have an installation count of more than one million on Google Play.

For Review Only

# Chapter 5

# Case Study

For Review Only

This chapter discusses the characteristics of a few apps taken from the 100 Indian financial apps, all of which were analyzed manually.

- **Yono Lite SBI - Mobile Banking (`com.sbi.SBIFreedomPlus`):** This application is for availing the internet banking facilities of the State Bank of India (SBI) [29]. This app is developed using the Kony framework [18]. We found that many parts of the code are initially encrypted and get loaded during runtime after appropriate decryption. By doing the native side analysis (i.e., analyzing the code which the developer has written in C/C++, which is the part of the `apk` as the shared object (*.so) file), we found a constant string of 64 characters in length. Via further inspection, we found this string to be the SHA-256 hash of the public key certificate used to sign the app. We modified the code on the native side (using radare2 [45]) to overwrite the value with our public key certificate, which will be used for signing the app after repackaging. When we launched this repackaged app, we could access the page where we had to enter the username and the password, indicating we had bypassed the first level of certificate check successfully. Without this step, we could not even reach the login page. But, post login, at a later point, we could no longer use this app, which suggests that there is some other anti-tampering check in the app that is not locatable by a keyword-based search.

- **BHIM App (`in.org.npci.upiapp`):** BHIM is a UPI-enabled initiative to facilitate safe, easy and instant digital payments through a mobile phone [10]. We observed that this app stopped working immediately in its repackaged version, and for the scenarios when the unmodified app was launched on a rooted device. While analyzing this app and inserting the logs at the appropriate location at the `smali` code, we found that the anti-tampering and other checks are at the native code level. But, even on the native side, we could not

locate the APIs corresponding to these checks by a keyword-based search, suggesting that static analysis alone cannot find the code associated with the SDC in this app.

- **APRB MobileBanking (`com.mbanking.aprb.aprb`):** This is the mobile banking application for the Arunachal Pradesh Rural Bank users [2]. We found 13 other apps have the same developer ID as this app in Google Play. The pattern in which the code is written for these SDCs is the same across all these apps. So, if an attacker can bypass the SDC in one app, they can bypass the SDCs in the remaining apps without any additional effort. We were able bypass the SDCs in the remaining 13 apps too.
- **BHIM Baroda Pay (`com.bankofbaroda.upi`):** This application is the UPI App from Bank of Baroda [11] and has all the SDCs. When we disassemble this app using apktool, a keyword-based search can locate the SDC related code. Our automation could bypass the anti-tampering check but could not bypass the other checks as this app uses SafetyNet API to perform device integrity checks. Since the validation is happening at the client side, we successfully bypassed the root detection and the emulator detection checks manually.

For Review Only

# Chapter 6

# Recommendations

Every SDC that an app developer inserts in an app is bypassable in principle; the question, though, is how hard it is for the attackers to bypass it. Here, we discuss guidelines that app developers should follow regarding the same. The SDC-related code should not be locatable via trivial static analysis (i.e., from keyword-based search). App developers should write the same on the native side of the app, as, compared to the Java level, bypassing the SDC at the native level is more challenging. Many existing frameworks help to put these checks at the native side of the app (e.g., Kony Framework [18]). Another technique is to use Google's SafetyNet API [48] accurately. The developer should send the JWS Object (a return value from this API; signed by Google server) to the app's backend server and correctly perform the validation there. If developers use the SafetyNet API correctly [47] and the code related to SafetyNet is not locatable by trivial static analysis, it would be hard for the attackers to bypass the checks.

For Review Only

# Chapter 7

# Limitations

For Review Only

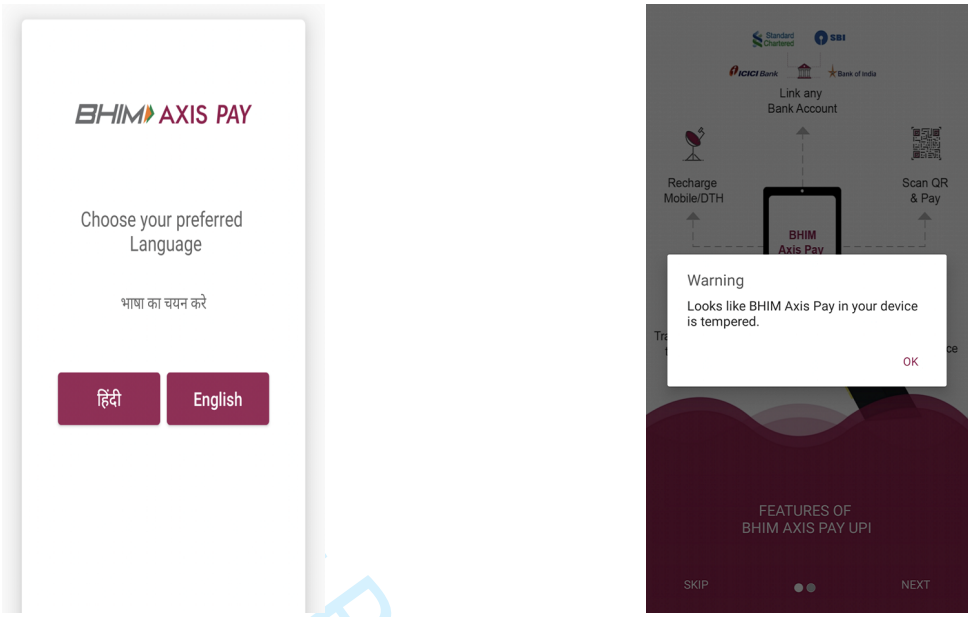
In this chapter, we discuss the limitations of our work with respect to how we detect the SDCs in apps, how we declare that an SDC has been bypassed, and why we cannot bypass the SDC for some apps.

Our methodology cannot bypass the SDC through automation if:

- The related code is either not locatable by a keyword-based search or present in the native part of the app.
- The app developer has used SafetyNet attestation.
- We cannot re-build the `apk` after `smali` code instrumentation due to the errors produced by the apktool. Mahmud et al. [64] have also excluded many apps from their dataset as they failed during reassembly due to errors in apktool.

We think that the SDC related code gets executed immediately after an app is launched. If this code gets executed at the later point of time (e.g., after user's successful login), our analysis could result in false negatives. Also, if the app displays an error message (regarding detection of compromised app/device) after some user interaction (e.g., after performing swipes, or clicking the "Next" button, etc.), our analysis cannot detect the check as the app's UI will be identical for both the runs until any user interaction (Figure 7.1). Similarly, if instrumented apps do not work at a later point of time, our approach will yield false results that we have successfully bypassed the check.

Consider a scenario when the app is launched on a compromised device. Even though it does not have the corresponding SDC, we may see a different UI compared with the original app's launch on the non-rooted device. It could be due to delay in the code execution in one of the runs (a probable reason could be a delay in response from app's backend server). If that is




(a) First Activity after launch. (b) Error message.

Figure 7.1: `com.upi.axispay` displays an error message after the selection of the language.

the case, our analysis could result in a false positive regarding SDC detection. We launched the app for a minute to ensure that a good amount of code was executed in both runs to account for such a possibility.

We also observed that a few apps display ads during their run. For those apps, due to such dynamic UI changes, we get different `xml` dumps (irrespective of the check present) when the app runs in the non-rooted real device and compromised device. Hence, we have not performed the entire `xml` comparison for the two runs, but instead checked the `resource-id` attribute of the `xml` dump to minimize the false positives.

The following are the drawbacks of UIAutomator that would have impacted our results:

- For the same app's UI, there is a difference between the UI elements fetched from `xml` dump when captured multiple times. For the sake of consistency, we consecutively captured the view hierarchy of the app's UI till the point when two consecutive `xml` dumps (i.e., the set of values retrieved from the `resource-id` attribute) are same.
- For some apps, it could not produce the app's view hierarchy because of not getting the ideal state of the app's UI (even after the launch for a minute). So apart from the UI analysis, we have done the log analysis to handle such scenarios.

Review Only

# Chapter 8

# Related Work

For Review Only

Many previous works have analyzed the presence of SDCs in Android apps and how they can be bypassed. Sun et al. [70] have developed a semi-automated tool RDAnalyzer to bypass the root detection checks via API hooking both in Java and native code. Nguyen Vu et al. [66] have also shown that many root detection checks can be evaded through API hooking and static file renaming. They had taken 18 financial banking apps of Korea and were able to bypass 10 out of 18 apps by changing the `su` binary filename to different name. Kim et al. [58] manually analyzed the 76 popular financial apps of the Republic of Korea. They bypassed 67 out of 73 apps that check device integrity and 39 out of 44 apps that check app integrity. Ibrahim et al. [57] performed the analysis on Google's SafetyNet API, and they manually bypassed the device integrity checks for 21 apps that use SafetyNet. However, all these studies that bypassed various checks were based on manual approaches; our work has tried to bypass these checks through automation.

Much has been accomplished in Android app repackaging and its countermeasures. Zhou et al. [77] found that 86% of the malware samples originate from repackaged versions of legitimate apps corrupted with a malicious payload. Their follow-up work [76] analyzed apps from the third-party Android marketplaces and found that 5% to 13% of the apps over there are repackaged. Earlier works [62],[71],[74] developed a mechanism that stops the working of a repackaged version of the original app on the real device. Merlo et al. [65] reviewed various anti-repackaging measures that have been proposed in the past. Berlato et al. [53] developed a static analysis tool, ATADetector, that detects anti-debugging and anti-tampering protection in their apps. However, they do not analyze the ease with which an attacker can bypass these checks.

Some research has also been done on the Android apps that provide payment services. Mahmud et al. [64] developed a static analysis tool, Cardpliance, to identify whether PCI DSS

(Payment Card Industry Data Security Standards) standards are being followed in the Android apps that ask the users to enter a credit card number. They found that only 1.67% of analyzed apps are not compliant with the PCI DSS. Z Din et al. [55] developed a system that other apps can integrate as an SDK that will help in preventing Card Not Present fraudulent transactions.

There has been a lot of research focused on financial apps. Reaves et al. [68] analyzed 46 well-known Android financial apps and found that majority of these apps fail to provide the necessary protection essential for financial services. Chen et al. [54] analyzed 693 banking apps across 83 countries and found 2,157 weaknesses on these apps. However, this work does not examine the self-defense checks recommended by OWASP. Kumar et al. [59] performed the security analysis of the UPI Protocol by manually analyzing and reverse engineering the BHIM App and confirmed their findings on other UPI Apps. Uddin et al. [72] developed a semi-automated security assessment framework, Horus, to analyze the crypto-wallet apps and validate the security standards critical for such apps. However, they also did not consider OWASP guidelines. Zungur et al. [78] analyzed financial apps to check if app developers are following guidelines recommended by OWASP to make their apps more secure. But, they do not identify the robustness of these checks.

Finally, UI analysis and log analysis techniques have been used by many other works in the past. Zhou et al. [75] found that UIAutomator can only capture the view hierarchy of the topmost focused window. Other works [69],[63],[61] have used the UIAutomator to capture the view hierarchies to find a particular view component. Zungur et al. [78] also detected the presence of the various checks in the app by analyzing the UI elements using UIAutomator. Girei et al. [56] proposed botnet detection techniques for mobile devices using log analysis. Yoo et al. [73] used the Logcat as a listener for automatic mobile app testing.

For Review Only

# Chapter 9

# Conclusion

This work focuses on the ease with which attackers can bypass the self-defense checks recommended by OWASP in Android apps. To the best of our knowledge, this is the first work that has performed the analysis at such a large-scale targeting 2,854 `FINANCE` category apps from Google Play and tried to bypass the checks through automation. Our work reveals that only 22.5% of the financial apps have all the SDCs and 27.1% of the financial apps have none of the SDCs we considered. Our work has bypassed the SDCs of some popular financial apps having installation count in the order of millions from Google Play. Furthermore, each of the self-defense checks we have considered in this work is present in only $< 50\%$ of the apps. With the help of various tools, this work successfully bypassed at least one self-defense check in more than 50% of the financial apps. We believe that this work will be beneficial for app developers to appreciate the value of the self-defense checks and insert these checks in a more non-bypassable manner.

For Review Only

# Appendix

## A Responsible Disclosure

In April 2022, we reach out to following banks whose banking apps do not have the SDCs, or the SDC has been bypassed:

- HDFC Bank
- SBI Bank
- Axis Bank
- Bank of Baroda
- Kotak Mahindra Bank
- ICICI Bank
- IDBI Bank
- Suryoday Small Finance Bank
- Arunachal Pradesh Rural Bank
- Amreli Jilla Madhyastha Sahakari Bank
- Bellary District Co-operative Central Bank
- Chikmagalur District Central Cooperative Bank
- Jila Sahakari Kendariya Bank Maryadit Durg
- Indore Premier Co-operative Bank
- Kodagu District Cooperative Central Bank

For Review Only

- Jila Sahakari Kendriya Bank Maryadit Khargone
- Kurmanchal Nagar Sahkari Bank
- Karnataka State Cooperative Apex Bank
- Mahalakshmi Co-operative Bank
- Mysore & Chamarajanagar District Co-operative Central Bank
- Mangalore Town Bank
- Vijayapura District Co-operative Central Bank
- Vyavsayik Sahakri Bank
- Prathama UP Grahmin Bank
- Dakshin Bihar Grahmin Bank
- Sarva Haryana Grahmin Bank
- Himachal Pradesh Grahmin Bank
- Punjab Grahmin Bank

We shared our results, including the technical details (with respect to how we bypassed the checks) and recommendations for the app developers to make the job really hard for the attackers to bypass such checks. Unfortunately, as of the date (29th April, 2022), only one bank have responded back to us.

For Review Only

# Bibliography

[1] Android Debug Bridge (adb). URL https://developer.android.com/studio/command-line/adb. 8

[2] APRB Mobile Banking. URL https://play.google.com/store/apps/details?id=com.mbanking.aprb.aprb. 28

[3] ActivityManager, . URL https://developer.android.com/reference/android/app/ActivityManager. 9

[4] ActivityTaskManager Source Code, . URL https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/ActivityTaskManager.java. 9

[5] Android Developers. URL https://developer.android.com/. 15

[6] Apksigner, . URL https://developer.android.com/studio/command-line/apksigner. 12, 14

[7] Apktool: A tool for reverse engineering android apk files, . URL https://ibotpeaches.github.io/Apktool/. 7, 14

[8] Google-Play-Signing, . URL https://developer.android.com/studio/publish/app-signing. 5

[9] Appium, . URL https://appium.io/. 8

[10] BHIM UPI App, . URL https://play.google.com/store/apps/details?id=in.org.npci.upiapp. 27

[11] BHIM Baroda Pay, . URL https://play.google.com/store/apps/details?id=com.bankofbaroda.upi. 28

For Review Only

# BIBLIOGRAPHY

[12] Build Source Code, . URL https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Build.java. 18

[13] Build — Android Developers, . URL https://developer.android.com/reference/android/os/Build. 6

[14] Frida. URL (https://frida.re/). 6

[15] Google-Play-Scrapper. URL https://github.com/facundoolano/google-play-scraper. 8

[16] PackageManager — Android Developers. URL https://developer.android.com/reference/android/content/pm/PackageManager#getInstallerPackageName(java.lang.String). 6

[17] Creating a KeyStore in JKS Format. URL https://docs.oracle.com/cd/E19509-01/820-3503/ggfen/index.html. 5, 12

[18] Kony Docs. URL https://docs.kony.com/7_3/konylibrary/visualizer/visualizer_user_guide/Content/ApplicationSecurity.htm. 27, 29

[19] Magisk App. URL https://github.com/topjohnwu/Magisk. 20

[20] NotificationManagerService Source Code. URL https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/notification/NotificationManagerService.java. 10

[21] Google Play Store. URL https://play.google.com/store. 2

[22] Selenium. URL (https://www.selenium.dev/). 8

[23] ShareIt India. URL https://yourstory.com/2018/10/creating-tribes-conquer-karam-malhotra-tells-shareits-india-story/amp?utm_pageloadtype=scroll. 7

[24] Signature Class Source Code. URL https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/Signature.java. 15

[25] Telephony Manager Android Developers. URL https://developer.android.com/reference/android/telephony/TelephonyManager. 6

For Review Only

# BIBLIOGRAPHY

[26] UI Automator. URL https://developer.android.com/training/testing/ui-automator. 3, 9

[27] Financial Apps US Statistics. URL https://www.emarketer.com/content/finance-apps-downloads. 1

[28] UI Automator Viewer. URL https://developer.android.com/training/testing/ui-automator#ui-automator-viewer. 8

[29] Yono Lite SBI - Mobile Banking. URL https://play.google.com/store/apps/details?id=com.sbi.SBIFreedomPlus. 27

[30] Android App Bundles. URL https://developer.android.com/guide/app-bundle. 12

[31] AAPT2. URL https://developer.android.com/studio/command-line/aapt2. 12

[32] Application Fundamentals, . URL https://developer.android.com/guide/components/fundamentals. 5

[33] State of Finance App Marketing, . URL https://www.appsflyer.com/resources/reports/finance-app-marketing-global/. 1

[34] ARM Apps on Emulator. URL https://developer.android.com/studio/releases/emulator#support_for_arm_binaries_on_android_9_and_11_system_images. 20

[35] Digital Wallets Popularity. URL https://www.financialexpress.com/industry/banking-finance/digital-wallets-emerge-second-most-popular-in-store-payment-method 2218021/. 1

[36] Dumpsys. URL https://developer.android.com/studio/command-line/dumpsys. 10

[37] Ghidra. URL https://ghidra-sre.org/. 7

[38] JADX - Dex to Java Decompiler. URL https://github.com/skylot/jadx. 7

[39] JAR File Specification. URL https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Signed_JAR_File. 15

[40] Logcat command-line tool. URL https://developer.android.com/studio/command-line/logcat. 3, 9

For Review Only

# BIBLIOGRAPHY

[41] OWASP Mobile Application Security Verification Standard. URL https://github.com/OWASP/owasp-masvs#owasp-mobile-application-security-verification-standard-masvs. 2, 15

[42] NPCI UPI. URL https://www.npci.org.in/what-we-do/upi/product-overview. 8

[43] OpenSSL. URL http://manpages.ubuntu.com/manpages/trusty/man1/openssl.1ssl.html. 15

[44] OWASP Mobile Top 10. URL https://owasp.org/www-project-mobile-top-10/. 1

[45] Radare2. URL https://github.com/radareorg/radare2. 7, 27

[46] Reserve Bank of India Annual Report. URL https://m.rbi.org.in/Scripts/AnnualReportPublications.aspx?Id=1322. 1

[47] SafetyNet Misuse, . URL https://android-developers.googleblog.com/2017/11/10-things-you-might-be-doing-wrong-when.html. 1, 29

[48] SafetyNet, . URL https://developer.android.com/training/safetynet. 1, 6, 29

[49] Toast Message. URL https://developer.android.com/guide/topics/ui/notifiers/toasts. 10

[50] APK Signature Scheme v1. URL https://source.android.com/security/apksigning#v1. 15

[51] APK Signature Scheme v2. URL https://source.android.com/security/apksigning/v2#verification. 5

[52] Xposed Framework. URL https://repo.xposed.info/module/de.robv.android.xposed.installer. 6, 20

[53] Stefano Berlato and Mariano Ceccato. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications*, 2020. 2, 17, 32

[54] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020. 33

For Review Only

# BIBLIOGRAPHY

[55] Zainul Abi Din, Hari Venugopalan, Jaime Park, Andy Li, Weisu Yin, HaoHui Mai, Yong Jae Lee, Steven Liu, and Samuel T. King. Boxer: Preventing fraud by scanning credit cards. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020. 33

[56] Daifur Abubakar Girei, Munam Ali Shah, and Muhammad Bilal Shahid. An enhanced botnet detection technique for mobile devices using log analysis. In *2016 22nd International Conference on Automation and Computing (ICAC)*, 2016. 33

[57] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. Safetynot: on the usage of the safetynet attestation api in android. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021. 2, 32

[58] Taehun Kim, Hyeonmin Ha, Seoyoon Choi, Jaeyeon Jung, and Byung-Gon Chun. Breaking ad-hoc runtime integrity protection mechanisms in android financial apps. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017. 2, 32

[59] Renuka Kumar, Sreesh Kishore, Hao Lu, and Atul Prakash. Security analysis of unified payments interface and payment apps in india. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020. 33

[60] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2021. 1

[61] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017. 33

[62] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. Repackage-proofing android apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016. 32

[63] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. Paladin: Automated generation of reproducible test cases for android apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, 2019. 33

[64] Samin Yaseer Mahmud, Akhil Acharya, Benjamin Andow, William Enck, and Bradley Reaves. Cardpliance: PCI DSS compliance of android applications. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020. 30, 32



For Review Only

# BIBLIOGRAPHY

[65] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. You shall not repackage! demystifying anti-repackaging on android. *Computers & Security*, 2021. 32

[66] Long Nguyen-Vu, Ngoc-Tu Chau, Seongeun Kang, Souhwan Jung, and Zonghua Zhang. Android rooting: An arms race between evasion and detection. *Sec. and Commun. Netw.*, 2017. 32

[67] Anh Pham, Italo Dacosta, Eleonora Losiouk, John Stephan, Kevin Huguenin, and Jean-Pierre Hubaux. HideMyApp: Hiding the presence of sensitive apps on android. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019. 14

[68] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015. 33

[69] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017. 33

[70] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015. 32

[71] Simon Tanner, Ilian Vogels, and Roger Wattenhofer. Protecting android apps from repackaging using native code. In *FPS*, 2019. 32

[72] Md Shahab Uddin, Mohammad Mannan, and Amr Youssef. Horus: A security assessment framework for android crypto wallets. In *International Conference on Security and Privacy in Communication Systems*, 2021. 33

[73] Hyunsik Yoo and Youngseok Lee. An automatic mobile app testing method with user event scenario. In *2017 18th IEEE International Conference on Mobile Data Management (MDM)*, 2017. 33

[74] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs. *IEEE Transactions on Dependable and Secure Computing*, 2021. 32



For Review Only

# BIBLIOGRAPHY

[75] Hao Zhou, Ting Chen, Haoyu Wang, Le Yu, Xiapu Luo, Ting Wang, and Wei Zhang. Ui obfuscation and its effects on automated ui analysis for android apps. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020. 33

[76] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, 2012. 7, 32

[77] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, 2012. 32

[78] Onur Zungur, Antonio Bianchi, Gianluca Stringhini, and Manuel Egele. Appjitsu: Investigating the resiliency of android applications. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021. 2, 9, 20, 25, 33

For Review Only